INFORMATION PROCESSING 89, G.X. Ritter (ed.)
Elsevier Science Publishers B.V. (North-Holland)

ZERO-DEFECT DESIGNS, WHY AND HOW: FORMAL VERIFICATION vs. AUTOMATED SYNTHESIS

Dominique BORRIONE

Laboratoire IMAG/ARTEMIS
B.P. 53X, F-38041 Grenoble Cedex, France

Paolo PRINETTO

Politecnico di Torino
Dipartimento di Automatica
I-10129 Turin, Italy

Invited Paper

Zero-defect design is a goal we must try to reach and CAD tools provide an increasingly important support to designers. High-level automated synthesis and formal verification are cooperating approaches to this end. This paper analyzes the general framework of digital design and the relationships between synthesis and verification as far as functional correctness is concerned, showing their limits, mutual dependencies, and how they can and should work together.

1 Introduction

No doubt that zero-defect VLSI design is not only a desire, but also a need: fast turn-around time, high redesign, fabrication, and testing costs, but also VLSI's increasing complexity and the need for a rapid exploration of several design alternatives foster research and investments in the field.

A lot of efforts have been spent in automating design, in validating process and manufacturing, and in product and field testing. This paper is restricted to the first issue, i.e., how VLSI can be designed to guarantee its functional correctness.

There are two avenues currently used to attack the problem: synthesizing correct by construction devices or verifying manual designs.

"*Synthesis is the process of mapping under some constraints an input specification for a hardware design into a hardware implementation, i.e., a multi-level network of realizable logic devices*" [Park84]. Some synthesis tools are already available on the market and their acceptance and use is growing in industrial environments. The first silicon compilers have been introduced on the market long ago [Sout83] and now some CAD vendors and most ASIC foundries have their own systems [JTBN86], [VZGa88]. A survey on techniques for logical synthesis can be found in [Newt85]. The domain of this paper is restricted to high-level synthesis, whose output may feed a logic synthesizer, i.e., a silicon compiler.

"*Verification is the process of showing that an implementation realizes its specification*". It may be based on the simulation of a (generally) non-exhaustive set of test cases or be a mathematical proof. The latter one is known in the literature as "**formal verification**". Advantages and disadvantages of both have been analyzed in [CaPr88]. Up to now there is no widespread consensus neither on methods nor on tools, which are mostly prototypical and not yet mature enough to be marketed. The potential benefits of formal verification are, however, raising the interest of industries, although this phenomenon may better be observed in Europe than in the US or in Japan.

From the point of view of design, synthesis should guarantee zero-defect products by means of correct by construction methods, whereas verification is a "post factum" process, mostly applied to individual cases, only. Correctness is related to the function a device is supposed to implement. If correctness by construction can be achieved, synthesis is superior to verification: as soon as automated tools provide results comparable, or better, than hand-made ones, verification should become useless. The bug is in the hypothesis: correctness is far from being guaranteed by current synthesis tools and that is where verification plays an important role. As reported in [BoDe88], none of the presenters at a recent workshop [1] claimed that the benchmark suite they synthesized was correct: several bugs had been found and attempts to simulate device behaviour yielded erroneous results. However, there is already a certain amount of efforts spent on correct by construction synthesis techniques, although they are sometimes restricted to particular domains. Some of the interdependencies between synthesis and verification have already been investigated in [Evek86]; in this paper we underline the need for verification within synthesis, outlining how both approaches can profitably cooperate toward zero-defect VLSI design.

Section 2 explores the Design Space, defining synthesis, verification, and correctness, and analyzes what kind of descriptions are needed in an integrated CAD environment. Then we survey on synthesis (section 3), paying particular attention to equivalence-preserving transformations and correct by construction methods. All this is ancillary to section 4, where we present the correlation between verification and synthesis.

2 The Design Space

To lay on firm basis the methods by which zero-defect VLSI

[1] ACM/IEEE High-level Synthesis Workshop, January 1988, Orcas Island, WA (USA)

design can be obtained, it is appropriate to turn to general design theory. According to the terminology of [Yosh80], "*designing is the act of creating an artificial object, which was previously non-existent in the real world, from some abstract concept obtained from knowledge about existing things*" The abstract concept input to the design process is called the "design specification" and the output (drawings, magnetic tapes, ...) from which the object can be manufactured are called the "design solution" or the "implementation".

2.1 Formalizing the Design Process

Both specification and implementation belong to multi-dimensional spaces, whose dimensions, called attributes, are heterogeneous in nature. The number of attributes is usually very large: a non-exhaustive list for specifications might include area, power consumption, speed, testability, functional behaviour, whereas attributes for implementations add to the above technology, transistor and pin count, placement and routing geometry, etc. Generally, the number of attributes in the implementation space I is larger than the number of attributes in the specification space S and some of the attributes of the former might not directly be attributes of the latter. We shall assume that the specification space S is a proper algebraic sub-space of the implementation space I, in the sense that it is possible to compute, for each design d of I, its value d_j, $j \in \{1 \ldots ns\}$ on each dimension of S. In order to evaluate both the severity of specifications and the acceptability of implementations, the only assumption is the existence of a partial pre-order $<_j$ on each dimension j of S. The partial pre-order is a total order for numeric attributes like area or cost.

We thus define a design function D: $S \rightarrow \mathcal{P}(I)$, where $\mathcal{P}(I)$ denotes the powerset of I, that puts into correspondence the two spaces: a design i is an implementation of a specification s if i belongs to $D(s)$. According to the terminology presented in [SiBa76] from a more restricted viewpoint, the set $D(s)$ of designs which implement a specification s is called the "design space" of s (Fig. 1).

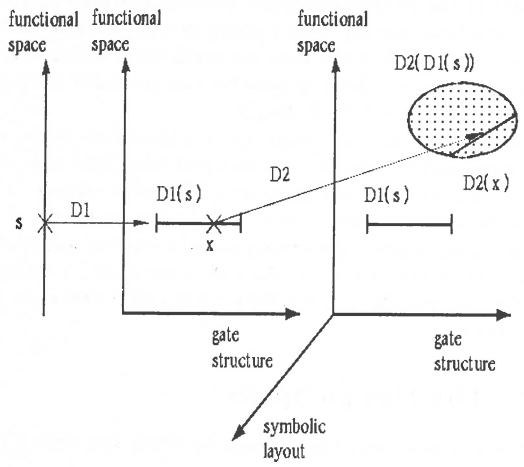

Figure 1: The Design Space

We shall call "validation function" the reverse of D: $V = D^{-1} : I \rightarrow \mathcal{P}(S)$. In general, function V is also a one-to-many application, since a design could very well satisfy two specifications, where one is less demanding or less precise than the other. We thus define a "satisfaction" partial pre-order Sat.

Sat can be induced on S, from the partial orders or pre-orders $<_j$ on the dimensions of S. If s_1 and s_2 are two specifications, s_1 satisfies s_2 if s_1 is lower than s_2 according to some attribute and s_1 is lower, equal, or not comparable with s_2 according to all the remaining attributes:

$$\forall\, s_1,\ s_2 \in S,\ s_1\ Sat\ s_2 \text{ iff}$$
$$\exists\, j \in \{1 \ldots ns\},\ \forall\, i \in \{1 \ldots ns\},\ i \neq j,$$
$$\neg proj_i(s_2) <_i proj_i(s_1)$$
$$\text{and } proj_j(s_1) <_j proj_j(s_2))$$

where ns is the number of dimensions of S and $proj_i$ is the projection of an element on the ith dimension.

Let d be a design in I, d_i, $i \in \{1 \ldots ns\}$, be the value of d on each dimension of S, and s_d be the specification such that

$$\forall\, j \in \{1 \ldots ns\}\ proj_j(s_d) = d_j$$

s_d is the most precise specification for d. In other words, design d is an implementation of all specifications s such that

$$s_d\ Sat\ s$$

Theorem:

s_1 and s_2 being two specifications, we have:

$$s_1\ Sat\ s_2 \text{ iff } D(s_1) \subseteq D(s_2)$$

Let Val be the single-valued function which associates to each element of I its most precise specification. In this framework, designing a circuit is the process of constructing an element d of I. The circuit implements a specification s if it can be shown that $d \in D(s)$.

In practice, a direct formulation of D as a whole cannot be given. It is easier to compute the validation function Val, which reduces the space dimensions and to show that $Val(d)\ Sat\ s$.

In addition, the designer must be able to evaluate and select among various elements of the design space. This corresponds to defining a partial pre-order P (for preference) on $D(s)$. For instance, between two circuits that satisfy a given functional specification, with the same area and speed, the one with better testability figures would be preferable; but if the better testable one were slower, may be the two circuits would be difficult to compare. This is why the preference relation is usually not a total pre-order.

Because of the large dimensions of the specification and implementation spaces, the design process is usually segmented into a series of steps, during which only a few attributes are being looked at, all the other attributes being considered as non-significant, or unchanged during the step. Resorting to information theory, [Koom78] calls "models" the specification and all intermediate partial implementations: a step is formalized as an information-adding transformation on mod-

els and the application of a stepwise design strategy reduces the total model complexity [Koom79].
As a corollary, the design function D may be expressed as the composition of the partial design functions D_p for the design steps, where most components of the partial functions are the identity:

$$D = D_n \circ D_{n-1} \circ \ldots D_1$$

As an illustration, let us consider the functional specification, the gate structure, and the symbolic layout as the three dimensions of the space. The process of designing a device that implements a function, which is a design step in a more complex process, is illustrated in Fig. 2.
$D_1(s)$ is the set of all gate structures which implement functional specification s. It can be viewed as a segment in the bidimensional space $Function \times Gate\_Structure$. For a particular gate structure x of this segment, $D_2(x)$ is the set of all possible symbolic layouts for x. $D_2(x)$ can be viewed as a segment in a tri-dimensional space

$$Function \times Gate\_Structure \times Symbolic\_Layout$$

whereas $D_2 \circ D_1(s)$ can be viewed as a plane subset in that space. Thus, the projection of a design element on a particular dimension represents the **semantics** of this element with respect to the dimension. For instance, since in Fig. 2 all the elements of $D_2(x)$ have the same projection on the functional space and on the gate structure, they share a common functional and structural semantics.

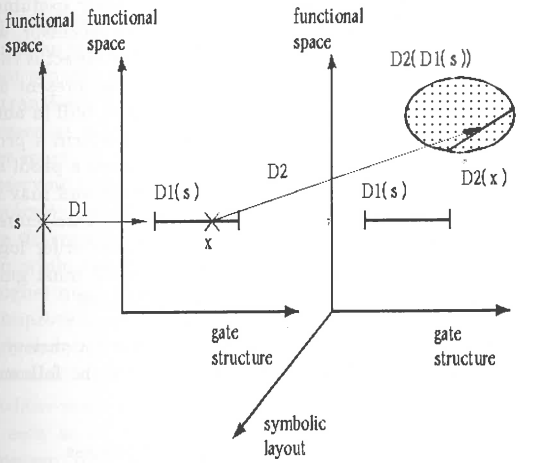

Figure 2: The Design Process

The above discussion may be related with other formalizations. For instance, in the ADAM project, the attributes of the design space are grouped into four independent subspaces: the behavior, structural, physical and timing/control subspaces, "*such that there are no implicit relationships between the objects of one subspace and the objects of another*" [KnPa85]. However, this structuring is seen from the point of view of design data management, where one design step corresponds to the hierarchical decomposition of an element into a set of more elementary objects within one of the four subspaces. A design element does not decompose into isomorphic hierarchies within the different subspaces, usually called "views" by CAD Data Base people [ChFo83].
We now have set all the necessary concepts to properly define the terminology used throughout this paper. In the following, we shall only consider the first VLSI design steps, where a key role is played by the behavioural specification attribute.
As a first approximation, let us note that the behaviour of a circuit at its outputs can be specified as a function of the inputs and possibly of the state variables. In [Gord80] it is shown that Denotational Semantics [Stoy77] offers a valuable theory for the formalization of complex hardware behaviour. It is not our intention to review this somewhat abrupt formalism, but rather to justify by this reference the fact that we can ultimately consider the circuit behavioural specification as an element in a functional space on which a partial pre-order is available.
We can now define synthesis as the process of obtaining an element of the implementation space from a functional specification. The synthesized circuit is error-free or zero-defect if it belongs to the Design Space of its specification. The goal of an automatic synthesis tool is to mechanically produce from a specification s an element of the design space $D(s)$, which should be as minimal as possible with respect to the preference relation P. Formal verification is the process of proving that an element of the implementation space indeed belongs to the design space.
Let d_1 and d_2 be two designs in the implementation space or two models resulting from the same design step. We say that d_2 is obtained from d_1 by application of an equivalence-preserving transformation if the projections of d_1 and d_2 on the functional space are equal. Equivalence-preserving transformations are applied, at any given design step, to find preferable designs/models.
Let $Satf$ now be the satisfaction partial pre-order, restricted to the functional space F ($Satf$ expresses the precision in functional specification). In essence, in a multi-step synthesis process, some design steps, going from a more abstract to a more detailed level of description of the circuit, add more precision to its functional semantics. This is formalized by the fact that, d_1 and d_2 being two successive partial implementations of specification s:

$$D(s) = D_n \circ D_{n-1} \circ \ldots D_1(s) \supseteq$$
$$D_n \circ D_{n-1} \circ \ldots D_2(d_1) \supseteq$$
$$D_n \circ D_{n-1} \circ \ldots D_3(d_2)$$

We say that d_2 is obtained from d_1 by application of a correctness-preserving transformation if the projection of d_2 on the functional space satisfies the projection of d_1:

$$Proj_F\,(Val(d_2))\; Satf\; Proj_F\,(Val(d_1))$$

2.2 Behavioral Specifications of Hardware

From the previous paragraphs, the reader will have inferred

that informal specifications, which have long been the only existing documents at the start of a project must rapidly be formalized, in order to be usable in a zero-defect computer-aided VLSI design process. Opinions may differ on the form, but the need for machine processable specifications is recognized as an absolute necessity. To be of practical value, machine processable specifications must fulfill a minimal set of requirements:

1. they must be sufficiently general for use at the various steps for a particular design and they should be applicable to a wide class of designs;
2. they must have unambiguous functional semantics;
3. they must be easy to interface to all other necessary design data;
4. there must exist a means to help the designer check that the specified behaviour is the intended behaviour.

With respect to this last point, it is not our intention to enter the philosophical debate: do specifications really express the will of their author and when should specifications cease to be specified by more specifications? We shall take the pragmatic attitude, recommended in [GHWi85], which consists in putting redundancy in specifications (possibly by saying the same thing using two different formalisms, or by stating expected properties of the specifications) and proving that no contradiction exists.
Specifications may be grouped in three broad classes:

1. **executable specifications** are a means to obtain output values, when input and internal state variables have been assigned a value. Executable specifications are typically written under the form of an algorithm. By executing his specifications on test inputs, the designer can gain some confidence that the specification adequately describes the intended circuit behaviour. Behavioral specifications, input to high level synthesis software, have been written in a variety of languages [Barb79], [CaRo85], [MMHK85]. Executable specification must be improved with additional information, such as pre/post-conditions and invariant properties [Floy67] [Hoar72] or algebraic axioms [DaHe85] that must be checked using the same techniques that have been developed for the formal verification of software [Muss80] [GHWi85].
2. **interpretable specifications** are written in a declarative or in a description language. They are non-algorithmic in the sense that the order in which the statements are written is not supposed to be essential in the computation of output values from input and internal state values (although it is often significant); some underlying operational semantics is assumed. Traditional Hardware Description Languages (HDLs) belong to this class and are the most popular machine processable specifications today. HDLs can be counted by dozens. Efforts to cover the specification needs of all design steps along the functional and structural dimensions have led to the emergence of multi-level HDLs; the first one with soundly defined semantics was CONLAN [PBBD83], whose concepts and definition strongly influenced (among the others) CASCADE [BoLe85], the most comprehensive language (it also includes detailed electrical behaviour modelling), and VHDL [IEEE88], the only standardized HDL to date.
3. **formally manipulable specifications** group a variety of approaches for modelling hardware behaviour, in terms of a formal system supported by mechanized reasoning. A review of current research efforts may be found in [CaPr88]. Formally manipulable specifications have not yet gone out of research and development laboratories. They have been written with the intention of proving mathematically that the result of a design step satisfies its behavioral specification. Opinions vary upon the formal system(s) best suited to describe and reason about hardware behavior. The great advantage of formally manipulable specifications lies in the fact that they can be directly used both to demonstrate facts (expected properties) about the specifications themselves, and to prove the correctness of a design step, provided its result is expressed in terms of the same formal system. Why then are formally manipulable specifications not yet in widespread use? Novelty is part of the answer: their usefulness on large circuits, such as a 16 bit micro-processor, was first demonstrated in [Hunt86]. Another aspect is their lack of user-friendliness and the fact that present day demonstrators require a great amount of skill in automatic theorem proving: the time to perform a proof and sometimes the mere fact of obtaining a proof depend on how the specification is written and may require the designer to introduce intermediate lemmas [CoPi88] in first order theories; in higher order logic the user of a tool such as HOL [Gord87] must guide the system through the proof steps himself.

Specifications in a CAD environment require a variety of software tools, which all contribute to one of the following tasks:

- going one step forward in the design process
- verification of a design step with respect to some attribute
- production of fabrication documents/tapes.

It is now well recognized that all these tools should be integrated around a common internal data structure. Data are produced from the compilation of user-provided specifications and information expressed in some language(s) (saying "language", we do not distinguish between graphic and textual syntax) and from the execution of programs such as synthesis tools and silicon compilers.
To restate the first requirement for a specification language, the same description should be used as input to as many design tools as possible; in other words, the specification language should be sufficiently general to be compiled on a

variety of design dimensions, thus expressing the semantics of the description with respect to the corresponding design attributes.
Some formally manipulable specifications may also be simulated [Miln86] and serve as an input to high level synthesis tools [ODon88]. Yet, despite the existence of implicit structure in formally manipulable specifications, explicit extraction of such a structure, in order to feed Fault Simulators or Automatic Test Pattern Generators still needs to be done. In addition, we strongly believe that, at each design step, the user should be allowed to retrieve all information under a human-readable form as pleasant, i.e., related to the design task under way, as possible.
For this reason, we believe that the burial date of conventional HDLs has not yet come. Rather, in order for mathematically sound methods to gain acceptance in the designers' community, automatic translation of HDLs and schematics in terms of appropriate theories should be included in an integrated CAD system. According to this viewpoint, first advocated within the framework of CONLAN [Evek85], formally manipulable specifications, under a demonstrator's input format, may become part of the internal data structure of a design.

3 Synthesis

Starting from the formal definition of synthesis given in Section 2, an intuitive interpretation leads to a top-down refinement process, where new details are added to a specification to transform it into an implementation. Synthesis may thus be seen as a *vertical* process moving depth-first in the design space. A similar, intuitive explanation of equivalence-preserving transformations may be given starting from the requirements arising during the design process. Given a design description, it is often very useful to alter the description while remaining at the same abstraction level. The designer may want to modify the output of a synthesis step to improve some of its attributes. It could be worthwhile to explore design alternatives, to optimize area and/or performance, or simply to add human contribution to amend the machine-made implementation. These processes can always be seen as augmenting the design with additional details. However, from the functional point of view, the level remains the same. If adding more details to function may be seen as a vertical process, equivalence-preserving transformations are often seen as horizontal tasks: the function is kept, i.e., the projections of the original and transformed designs are the same in the functional space (see section 2), subsidiary attributes are changed.
The following subsections elaborate on these topics with particular attention to the correctness problem. It is not within the scope of this paper to be an introduction to the field, thus please refer to [Park84] and [MFPC88] as tutorials.

3.1 The basic ideas

High-level synthesis systems may be loosely grouped in two categories [Park84], according to the level of abstraction of the specification and of the implementation:

- behavior: synthesis of control-flow or abstract behavior;
- register-transfer: synthesis of register-transfer structure from abstract behavior, control-flow behavior, or register-transfer behavior.

The synthesis process is composed of distinct but non-independent subtasks whose activity is further complicated by the dual presence of goals and constraints. Constraints are imposed a priori by the designer to judge whether the synthesized results are satisfying, the goals are implicit in the design process. Constraints may be generally seen as imposing a higher bound on the preference relation P (see section 2) to guarantee acceptance. They range among the following [Park84]: speed, cost (area, package count, pin count, power consumption), design turn-around time, reliability, and testability.
The synthesis process is commonly subdivided into the following subproblems [MFPC88]:

- compilation: translation from human-readable form into machine-readable form, possibly with optimizations and checks;
- resource allocation: selection of structures to implement functions (many-to-many mapping), minimizing the amount of hardware needed;
- design transformation: changing a design to achieve a goal or satisfy a constraint without adding more details;
- decomposition: simplifying the function until direct allocation may occur to components of an available library (module binding)
- event scheduling: operation assignment to time slots, minimizing global time and/or control steps.

A detailed analysis of the algorithms which are currently used to solve the subproblems may be found in [MFPC88].

3.2 Correct by construction techniques

The issue of correctness has always been deeply felt by people working on synthesis, but only a restricted number of approaches addressed it. We can investigate these formal efforts classifying them as synthesis or equivalence-preserving transformations.
In general, equivalence-preserving transformations are used to optimize existing designs, according to the goals and constraints seen in the previous section. In [MFPa83] an abstract model of hardware behaviour is presented, based on "*behaviour expressions*". The model is mathematically sound, based on regular expressions augmented by predicates, and a number of optimizing transformations have been derived within it. These transformations have been used in the Carnegie-Mellon Design Automation system's internal form VT (Value Trace) [MFar81].
Correct-by-construction synthesis has been until now restricted to very special domains, such as combinational networks, systolic arrays, finite state machines, and multipliers. There is a trade-off between the generality of application and

the ability to derive formally proven synthesizers.
In [KaWo85] theorem-proving techniques are adopted to synthesize combinational logic. The elements of the design process are represented as a set of axioms in a theory, the problem of realizability of the function is the theorem, and a theorem-prover is used to demonstrate its validity within the theory. Additional procedures are then used to derive the actual implementation from the proven theorem.
In [PrLi88] a combined methodology for the specification of abstract synchronous data types and the proof of their systolic implementation is provided. The specification is based on an extension of the Parnas trace method [BaPa78] for software modules. This extension naturally possesses properties which make it suitable for systolic VLSI implementation. Systematic proof techniques are also derived to establish the correctness of several novel systolic implementation of well-known data types.
Finite state machines may be synthesized starting from their Glushkov automaton by some correctness-preserving transformations based on P-functions [SnTh86]. P-functions may be considered as a generalization of switching theory.
An interesting cross-fertilization between formal verification and synthesis is reported in [ChGr88]. Higher-ordel Logic, the HOL proof checker [Gord87], and the Scheme functional language [ReCl86] cooperate to describe and prove synthesis functions for Pezaris-like array multipliers. Synthesis functions preserve functional correctness.

4 Synergy Verification/Synthesis

The correct-by-construction approaches and the equivalence-preserving transformations are novel directions for investigation. Unluckily, their applicability domain is restricted to very special cases, as seen in the previous section. This is why there should be a synergy between verification and synthesis and this section elaborates on this topic.
Formal verification techniques can help in many ways the synthesis process. A first application is represented by the single subtasks in which synthesis may be partitioned. If the subtask consists of an algorithm whose function may be completely specified, then it is possible to investigate how this algorithm can be proven correct. The expression of a function realized by an algorithm is not always feasible: sometimes what is useful is to prove a partial correctness of the algorithm with respect to some significant properties. When the synthesis approach is based on Artificial Intelligence and its knowledge is concentrated in a knowledge-base, particular care must be paid to the problem of coherence and completeness.
Beside these promising research fields, it is highly probable that complete verification of a synthesis system is not feasible and a 90% verified synthesizer is not very useful, because the bugs in the remaining 10% will cause the crash. Post factum verification therefore remains an unavoidable burden: the synthesizer can process the specifications, but the verification tool must check the results until a sufficient confidence may be felt in the goodness of the approach. Moreover, whenever little changes are manually made on a synthesized design, verification is needed to guarantee the compliance with the original specifications.
Verification of post factum designs is also necessary for highly-critical systems, i.e., systems where the functional correctness confidence must be as high as possible. Whenever this is needed, verification and synthesis should rely on different methodologies, in order to avoid introducing the same kind of bugs. In the general case, as seen in the previous section and as reported in [Evek86], both synthesis and verification may profit from the development of the same methods and tools.
There is one more aspect to be taken into consideration: formally verifying a design or a synthesis tool is in itself a very difficult task, especially the second one, but the proof only guarantees that the method is correct, but says nothing as to what the mechanization does. The proof could thus be correct in theory, but the software realizing it may not be bug-free. Caveat: this statement should not be taken from a "philosophical" point of view: otherwise there should a proof, a prover, a proof of the prover, a prover for proofs of provers, and so on ad infinitum. There are, however, some practical implications: first, the need for software verification techniques, second the importance of cross-fertilizing once more synthesis and verification: the post factum verifier could discover bugs due to the synthesis tool's software. There is a growing awareness of this need [Park84], witnessed by the use of the verifier within the DDL system to eliminate the effect of the bugs in the simulator [Ueha86].

5 Conclusions

Rather than being a "positive" paper, claiming that great performances of synthesis or verification tools are already reality, this article proposes a critical view of the state-of-the-art techniques in this field. We pointed out that the main goal of synthesis is functional correctness and we have shown that very few efforts have been done up to now to achieve it. No solutions have been presented, rather some directions along which investigation may lead to good results have been indicated. They all concern the synergy between verification and synthesis techniques.
Unluckily, the complexity of the design space seems to be so enormous that, unless some restrictions are imposed, no considerable results will ever be reached at a reasonable cost. We thus advocate the development of "Design for Verifiability" methodologies, just as people did for testability and their inclusion in synthesis tools. Although much remains to be made, following the parallel and interleaved paths of verification and synthesis may prove to be the right way to approach zero-defect VLSI design.

6 Acknowledgments

We would like to thank Dr. Paolo Camurati for his valuable help in preparing this paper.

7 References

[**BaPa78**] W. Bartussek, D.L. Parnas: "Using assertions about traces to write abstract specification for software modules," 2nd Conf. Eur. Coop. Inform., Berlin (FRG), Springer Verlag, 1978

[**Barb79**] M. R. Barbacci: "Instruction Set Processor Specifications for Simulation, Evaluation, and Synthesis," DAC-16: 16th ACM/IEEE Design Automation Conference, San Diego CA (USA), June 1979, pp. 64-72

[**BoDe88**] G. Borriello, E. Detjens: "High-level Synthesis: Current Status and Future Directions," DAC-25: 25th ACM/IEEE Design Automation Conference, Anaheim, CA (USA), June 1988, pp. 477-482

[**BoLe85**] D. Borrione, C. Le Faou: "Overview of the CASCADE Multi-level Hardware Description Language and its Mixed-Mode Simulation Mechanisms," CHDL'85: IFIP 7th Int. Symposium on Computer Hardware Description Languages and their Applications, Tokyo (Japan), August 1985, pp. 239-260

[**CaPr88**] P. Camurati, P. Prinetto: "Formal verification of hardware correctness: introduction and survey of current research," IEEE Computer, Vol. 21, n. 7, July 1988, pp. 8-19

[**CaRo85**] R. Camposano, W. Rosenstiel: "A design environment for the synthesis of integrated circuits," 11th EUROMICRO Symposium: "Microcomputers, Usage and Design", Brussels (Belgium), September 1985, pp. 211-215

[**ChFo83**] L. Cholvy, J. Foisseau: "Representation of Information in a Design Process," CAPE'83: Intl. Conference "Computer Applications in Production and Engineering", North Holland, 1983, pp. 545-557

[**ChGr88**] S.K. Chin, K. Greene: "Verifiable and Executable Theories of Design for Synthesizing Correct Hardware," ICCD '88: IEEE International Conference on Computer Design: VLSI in Computers & Processors, Rye Brook, NY (USA), October 1988, pp. 604-610

[**CoPi88**] H. Collavizza, L. Pierre: "Formal Verification of Hardware using OBJ and the Boyer-Moore Theorem Prover: a Practical Comparison," Research Report n. 88-04, Laboratoire MAIUP, Marseille (France), October 1988

[**DaHe85**] S. Dasgupta, J. Heinanen: "On the axiomatic specification of computer architectures," CHDL'85: IFIP 7th Int. Symposium on Computer Hardware Description Languages and their Applications, Tokyo (Japan), August 1985, pp. 1-13

[**Darr79**] J. Darringer: "The application of Program Verification Techniques to Hardware Verification," DAC-16: 16th ACM/IEEE Design Automation Conference, San Diego CA (USA), June 1979, pp. 375-381

[**Evek85**] H. Eveking: "The application of CHDL's to the abstract specification of hardware," CHDL'85: IFIP 7th Int. Symposium on Computer Hardware Description Languages and their Applications, Tokyo (Japan), August 1985, pp. 167-178

[**Evek86**] H. Eveking: "Verification, synthesis and correctness-preserving transformations - cooperative approaches to correct hardware design," IFIP WG 10.2 Workshop "From HDL descriptions to guaranteed correct circuit designs", Grenoble (France), September 1986, pp. 207-217

[**Floy67**] R. W. Floyd: "Assigning meanings to programs," Symp. Mathematical Aspects of Computer Science, J.T. Schwartz ed, American Mathematical Society 1967, pp. 19-32

[**GHWi85**] J. Guttag, J. Horning, J. Wing: "The Larch Family of Specification Languages," IEEE Software, September 1985, pp. 24-36

[**Gord80**] M. Gordon: "The Denotational Semantics of Sequential Machines," Information Processing Letters, Vol. 10, N.1, February 1980

[**Gord87**] M. Gordon: "A proof generating system for Higher-Order Logic," VLSI verification, specification, and synthesis, G. Birtwistle and P.A. Subramanyam editors, Kluwer, 1987

[**GuHo78**] J.V. Guttag, J.J. Horning: "The Algebraic Specification of Abstract Data Types," Acta Informatica 10, 1978, pp. 27-52

[**Hoar72**] C. Hoare: "Proof of correctness of data representations," Acta Informatica, Vol.1, 1972, pp. 271-281

[**Hunt86**] W.A. Hunt: "FM8501: a verified microprocessor," IFIP WG 10.2 Workshop "From HDL descriptions to guaranteed correct circuit designs", Grenoble (France), September 1986, pp. 85-114, North-Holland, Amsterdam (Holland)

[**IEEE88**] IEEE: "Standard VHDL Language Reference Manual," IEEE Tsd 1076-1987, March 1988

[**JTBN86**] W. Joyner, L.H. Trevillyan, D. Brand, T.A. Nix, S.C. Gundersen: "Technology Adaptation in Logic Synthesis," DAC-23: 23th ACM/IEEE Design Automation Conference, Las Vegas, NE (USA), June 1986, pp. 94-100

[**KaWo85**] W.C. Kabat, A.S. Wojcik: "Automated synthesis of combinational logic using theorem-proving techniques, " IEEE Transactions on Computers, Vol. C-34, n. 7, July 1985, pp. 610-632

[**KnPa85**] D.W. Knapp, A.C. Parker: "A Unified Representation for Design Information," CHDL'85: IFIP 7th Int. Symposium on Computer Hardware Description Languages and their Applications, Tokyo (Japan), August 1985, pp. 337-353

[**Koom78**] C.J. Koomen: "Information Laws for System Design," International Conference on Cybernetics and Society, Tokyo-Kyoto (Japan), November 1978

[**Koom79**] C.J. Koomen: "Reducing model complexity in system design," International Conference on Cybernetics and Society, Denver, CO (USA), October 1979

[**Miln86**] G. Milne: "Towards verifiably correct VLSI design," in: "Formal Aspects of VLSI Design, Proc. Workshop on VLSI, Edinburgh (UK), 1985, ed. G. Milne and P. Subrahmanyam, North Holland (1986)

[**MFar81**] M.C. McFarland: "Mathematical models for formal verification in a design automation system," Technical Report, Design Research Center, Carnegie-Mellon University, Pittsburg PA (USA), 1981

[**MFPa83**] M.C. McFarland, A.C. Parker: "An Abstract Model of Behavior for Hardware Descriptions," IEEE Transactions on Computers, Vol. C-32, n. 7, July 1983, pp. 621-637

[**MFPC88**] M.C. Mc Farland, A.C. Parker, R. Camposano: "Tutorial on High Level Synthesis," DAC-25: 25th ACM/-IEEE Design Automation Conference, Anaheim CA (USA), June 1988, pp. 330-336

[**MMHK85**] T. Mano, F. Maruyama, K. Hayashi, T. Kakuda: "OCCAM to CMOS: Experimental Logic Design Support System," CHDL'85: IFIP 7th Int. Symposium on Computer Hardware Description Languages and their Applications, Tokyo (Japan), August 1985, pp. 381-390

[**Muss80**] D. Musser: "Abstract Data Type Specification in the Affirm System," IEEE transactions on Software Engineering, Vol. SE-1, n.1, 1980, pp. 24-32

[**Newt85**] A.R. Newton: "Techniques for Logic Synthesis," VLSI'85: IFIP WG 10.5 International Conference on Very Large Scale Integration, Tokyo (Japan), August 1985, pp. 27-43

[**ODon88**] J. O'Donnell: "Hydra: Hardware Description with Recursion Equations and Higher Order Combining Forms," IFIP WG 10.2 Workshop "The fusion of hardware design and verification", Glasgow (UK), July 1988, pp. 305-324

[**Park84**] A.C. Parker: "Automated Synthesis of Digital Systems," IEEE Design & Test, November 1984, pp. 75-81

[**PBBD83**] R. Piloty, M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill, P. Skelly: "CONLAN Report," Lecture Notes in Computer Science 151, Springer Verlag, Berlin (FRG), 1983

[**PiSt82**] V. Pitchumani, E. Stabler: "A formal method for computer design verificaiton," DAC-19: 19th ACM/IEEE Design Automation Conference, Las Vegas, NV (USA), June 1982, pp. 809-814

[**PrLi88**] D.K. Probst, H.F. Li: "Abstract Specification of synchronous data types for VLSI and proving the corectness of systolic network implementations," IEEE Transactions on Computers, Vol. C-37, n. 6, June 1988, pp. 710-720

[**ReCl86**] J. Rees, W. Clinger: "Revised report on the algorithmic language Scheme," MIT Artificial Intelligence Lab., AI Memo 848A, September 1986

[**SiBa76**] D.P. Siewiorek, M.R. Barbacci: "The CMU RT-CAD system - An innovative approch to computer aided design," AFIPS National Computer Conference 1976, pp. 643-655

[**SnTh86**] D. Snyers, A. Thayse: "Algorithmic State Machine Design and Automatic Theorem Proving: two dual approaches to the same activity," IEEE Transactions on Computers, Vol. C-35, n. 10, October 1986, pp. 853-861

[**Sout83**] J.R. Southard: "MacPitts: an approach to silicon compilation," IEEE Computer, December 1983, pp. 74-82

[**Stoy77**] J.E. Stoy: "Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory," MIT Press, 1977

[**Subr83**] P.A. Subrahmanyam: "Synthesizing VLSI Circuits from Behavioral Specifications: A Very High Level Silicon Compiler and its Theoretical Basis," VLSI'83: IFIP TC 10/WG International Conference on Very Large Scale Integration, Trondheim (Norway), August 1983, pp. 195-210

[**Ueha86**] T. Uehara: "Proof and synthesis are cooperative approaches for correct circuit designs," IFIP WG 10.2 Workshop "From HDL descriptions to guaranteed correct circuit designs", Grenoble (France), September 1986, pp. 219-230

[**VZGa88**] N. Van der Zanden, D. Gajski: "MILO: A Microarchitecture and Logic Optimizer," DAC-25: 25th ACM/-IEEE Design Automation Conference, Anaheim CA (USA), June 1988, pp. 403-408

[**Yosh80**] H. Yoshikawa: "General Design Theory and a CAD system," IFIP WG 5.2 Working Conference "Man-machine Communication in CAD/CAM", Tokyo (Japan), October 1980